Static Analysis to Identify Invariants in RSML Specifications*

David Y.W. Park, Jens U. Skakkebæk, and David L. Dill

Computer Science Department, Gates Building 3A
Stanford University, Stanford, CA 94305, USA.
E-mail: {parkit,jus,dill}@cs.stanford.edu

Abstract. Static analysis of formal, high-level specifications of safety critical software can discover flaws in the specification that would escape conventional syntactic and semantic analysis. As an example, specifications written in the Requirements State Machine Language (RSML) should be checked for *consistency*: two transitions out of the same state that are triggered by the same event should have mutually exclusive guarding conditions. The check uses only behavioral information that is *local* to a small set of states and transitions.
However, since only local behavior is analyzed, information about the behavior of the surrounding system is missing. The check may consequently produce counterexamples for state combinations that are not possible when the behavior of the whole system is taken into account. A solution is to identify invariants of the global system that can be used to exclude the impossible state combinations. Manually deriving invariants from designs of realistic size is laborious and error-prone. Finding them by mechanically enumerating the state space is computationally infeasible. The challenge is to find *approximate* methods that can find fewer but adequate invariants from abstracted models of specifications.
We present an algorithm for deriving invariants that are used to exclude impossible counterexamples resulting from checking consistency of transitions in RSML. The algorithm has been implemented in an RSML prototype tool and has been applied successfully to the static checking of version 6.04a of the (air) Traffic alert and Collision Avoidance System (TCAS II) specification.

1 Introduction

Formal, high-level specifications of safety critical software are being advocated to reveal flaws in software early in the design phase [3,8,10,12]. In contrast to informal specifications, formal specifications can be checked for wellformedness beyond trivial syntactic properties [1,6,7,11]. For instance, specifications written in the Requirements State Machine Language (RSML) [10] should be checked to ensure that the specification is *consistent* [9]: two transitions out of the same state that are triggered by the same event should have mutually exclusive guarding conditions. An inconsistency inadvertently allows for several different implementations, which may complicate testing, verification, and reuse of the software.

* The research was supported by the Defense Advanced Research Projects Agency under contract number DABT63-96-C-0097-P00002.

Checking consistency using model checking is infeasible for designs of realistic size [6]. Instead, the guarding conditions can be converted into logical predicates and checked for mutual exclusion. We have specifically used the Stanford Validity Checker (SVC) [2], a decision procedure for a quantifier-free fragment of first-order logic [11]. The check is more efficient than model checking, since it uses only behavioral information that is *local* to a small set of states and transitions.

However, since only local behavior is analyzed, information about the behavior of the surrounding system is missing. The check may consequently produce counterexamples for state combinations that are not possible when the behavior of the whole system is taken into account. For example, a purely local check may report that two transitions can be enabled simultaneously whenever one component state machine is in state $s1$ and another is in state $s2$. However, a more global analysis might reveal that this combination of circumstances can not occur, indicating that the local check has reported a non-existent problem. A solution is to identify invariants of the global state that can be used to exclude some of the impossible state combinations. Manually deriving invariants from designs of realistic size is laborious and error-prone. Finding them by mechanically enumerating the state space is computationally infeasible.

The solution we propose is to find *approximate* methods that can find fewer but still sufficient invariants from abstracted models of specifications. Significant size reductions can be achieved by omitting information in the abstraction process. We present an algorithm for deriving invariants that rule out some of the impossible counterexamples when checking consistency in RSML. The algorithm has been integrated in an RSML prototype analysis tool and has been applied successfully to the static checking of part of version 6.04a of the specification of the (air) Traffic alert and Collision Avoidance System (TCAS II) [10]. It is likely that the algorithm can be generalized to other variations of statecharts [4,5].

2 Motivating Example

We illustrate our approach with an example. An RSML Component State Machine (CSM), shown in Figure 1, consists of a set of input variables, a hierarchical state machine, and a set of output variables. When an external event arrives at the boundary of the CSM, the state machine executes using the values of the input variables, assigning new values to the output variables.

As in statecharts, individual states may themselves contain state machines. A state is *active* if control resides in that state, and *inactive* otherwise. The predicate *in*(s) means that state s is active. State *Root* is of type *and*, so its immediate substates A, B, C, D, and E (outlined by dotted lines) must be active simultaneously if *Root* is active. State A is of type *or*, so at most one of its immediate substates may be active. A *basic* state has no children.

A transition is annotated with *trigger [guard]/output-event* and is taken if and only if its guarding condition is true when its trigger event occurs. If taken, the transition may generate an output event that triggers other transitions in the CSM. The guarding conditions are logical expressions over the values of the input variables and the active/inactive status of the states inside the CSM.

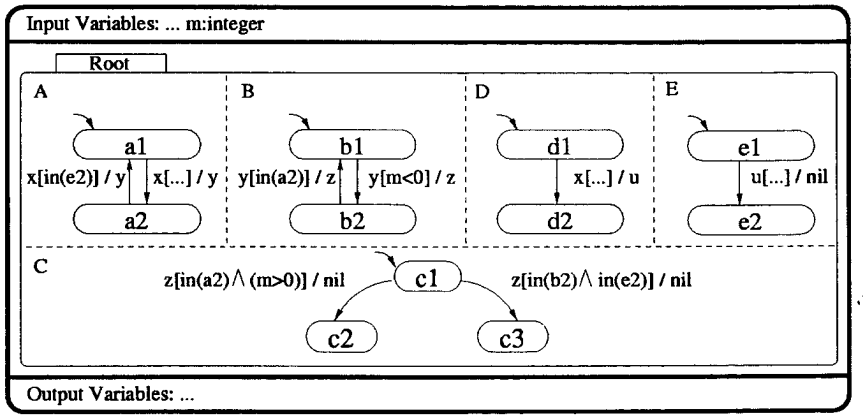

Fig. 1. An RSML component state machine to be checked for consistency.

The CSM executes in ***supersteps***. A superstep begins when an external event arrives at the boundary of the CSM. The external event begins a chain of internally generated events that trigger transitions within the CSM. In our example, external event x triggers transitions in states A and D in parallel. If taken, these transitions trigger transitions in states B and E. The transition in state B may, in turn, trigger transitions in state C, concluding the superstep. The event ordering scheme is shown in Figure 2(a).

Transitions in the CSM are ***consistent*** if and only if every pair of transitions out of the same state with the same trigger event has guarding conditions that can not be enabled simultaneously. For instance, transitions c1→c2 and c1→c3 are inconsistent under the condition $in(b2) \wedge in(e2) \wedge in(a2) \wedge (m > 0)$ since both guarding conditions are satisfied. Thus, the local check indicates that the transitions are potentially inconsistent. In such a situation, we say that the transitions are ***locally inconsistent***.

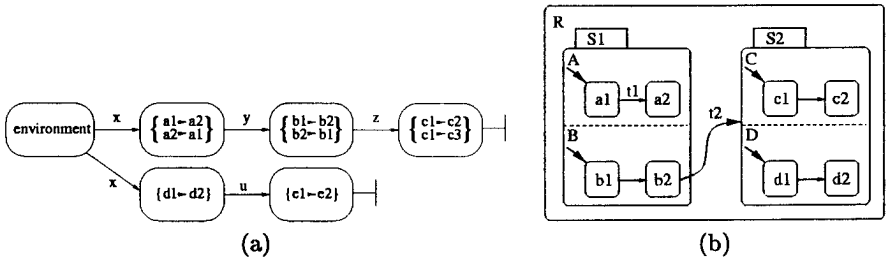

Fig. 2. (a) Event flow in the CSM. (b) Illustration of source and target completions.

However, a static analysis of a superstep can show that $in(a2)$ and $in(b2)$ can not be true at the same time, invalidating the inconsistency condition. Assume that the inconsistency condition holds when the two transitions in state C are triggered. Given the event tree in Figure 2(a), this assumption implies the following. First, predicate $(m > 0)$ must be true from the beginning of the superstep

since it is an input variable. Second, predicate $in(a2)$ was either true from the beginning of the superstep, or it recently became true if transition a1→a2 was taken. In either case, predicate $in(a2)$ must be true after transitions in state A are triggered since this is the last opportunity in which it can be made true before transitions in state C are triggered. Similarly, predicate $in(e2)$ must be true after transitions in state E are evaluated.

Finally, predicate $in(b2)$ must be true after state B evaluates. However, a stronger claim can be made since the guarding condition of transition b1→b2 is $[(m < 0)]$. Since predicate $(m > 0)$ must be true from the beginning of the step, this transition could not have been enabled and thus predicate $in(b2)$ must also have been true from the beginning of the superstep.

From the upper branch of the event tree in Figure 2(a) we know that transitions in state A evaluate before those in B. Therefore, $in(a2)$ must be true when transitions in state B are triggered. But this means that transition b2→b1 with guarding condition $[in(a2)]$ is enabled, making $b2$ no longer active. This contradicts the requirement that $in(b2)$ must be true from the beginning of the superstep. The inconsistency condition is therefore invalidated, and we derive the invariant: $(in(a2) \wedge (m > 0)) \Rightarrow \neg in(b2)$.

3 RSML

An RSML specification is a collection of CSMs which communicate through asynchronous message passing. Refer to [10] for a comprehensive description of the syntax and semantics. We focus on the internal behavior within a CSM and introduce concepts used later in the explanation of the approach.

3.1 Transitions

The explicit source and destination states of a transition are the states connected by the tail and head of the transition arrow. In Figure 2(b), the explicit source state and target state of transition $t2$ are $b2$ and $S2$ respectively.

Due to the hierarchical nature of the states, the explicit source and target states may not be the only states that are left and entered on a transition. In Figure 2(b), transition $t2$ not only leaves state $b2$, it also leaves $S1$ and all of its substates. This is because state $S1$ can not be active when the explicit target state $S2$ is active (they are both children of an *or* state). Similarly, state $S2$ is not the only state that is entered. $S2$ is an *and* state, so states C and D are also entered. Since C is an *or* state and no child state is specified to be the target, we enter state $c1$ by default. Default states are indicated by a transition arrow without a source. Likewise, state $d1$ is entered by default.

The set of all states that can not be active after taking a transition t is denoted *source-completion*(t) and the set of all states that may become active is denoted *target-completion*(t). Both sets can be determined statically. Informally, *source-completion*(t) is the set of all substates of the highest level state exited on the transition, and *target-completion*(t) is the set of default substates and explicitly targeted substates of the highest level state entered.

Identity transitions may be specified, although they are not shown in the CSM diagram. They are taken when no other transition out of the state is

enabled. By the RSML semantics used in TCAS II, identity transitions do not cause any state changes, and their sole purpose is to propagate trigger events.

3.2 Behavior of the CSM

A *superstep* takes the CSM from one global state to the next, where a global state is represented by the values of variables and the set of active states in the hierarchical state machine. A superstep is decomposed into a series of *microsteps*. A microstep can intuitively be thought of as a wavefront of transitions that are taken concurrently, in an arbitrary interleaving. The transitions in each microstep generate the set of events that trigger the transitions in the subsequent microstep. Transitions in a microstep are evaluated only after all transitions in the preceding microstep have been evaluated. An external trigger event from the environment begins the first microstep. The superstep ends when there is a microstep in which no more transitions are triggered.

4 Overview of the Algorithm

Given a local inconsistency condition, we look for an invariant that shows that the condition is unsatisfiable. Since this condition is a conjunction of predicates, it suffices to show that at least one predicate fails to hold, given the others.

First, the behavior of the CSM is abstracted, resulting in a model delineating which transitions can be triggered at each microstep. In *Backward Pass*, we begin by assuming that the local inconsistency condition holds at the last microstep (the microstep in which the locally inconsistent transitions are triggered). We then determine the earliest microstep from which each predicate must hold if it is to hold at the last microstep. In *Forward Pass*, we try to establish a contradiction by showing that some predicate in the inconsistency condition can not hold in the last microstep given other predicates determined to hold from prior microsteps. An invariant is formulated from the results of the analysis.

5 The Causality Model

The behavior of the CSM is abstracted as a model called the *causality tree* that delineates which transitions can be triggered at each microstep. Figure 3a is the causality tree for the superstep initiated by external event x in the CSM from Section 2. A node in the tree represents a set of transitions with the same input and output triggers. The directed edge into a node represents the trigger event to transitions associated with the node, and the directed edge out of the node represents the output event (possibly empty).

Beginning with the external trigger event from the environment node, we add nodes containing transitions triggered by the event. These new nodes may have their own output triggers which become directed edges to subsequent nodes with transitions triggered by them. Nodes are added until all leaves of the tree have the empty output trigger. Circularities in the event propagation are not allowed, since they lead to infinite paths in the causality tree. The algorithm trivially checks for circularities each time a new node is added, and aborts the tree construction if a circularity is detected.

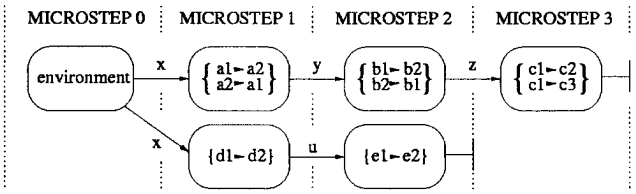

Fig. 3. Causality tree for the CSM in Figure 1.

The depth of a node in the tree denotes the microstep number in which its transitions may be triggered. Hence, grouping all of the nodes in the tree by their depth effectively determines the set of those and only those transitions that can be triggered in each microstep. Transitions in states B and E, for instance, may be triggered simultaneously in microstep 2. Note that the causality tree is a conservative model: it captures what events *may* be generated at each microstep, without information about whether guarding conditions are enabled. Identity transitions are not included in the model since they are not important to our analysis. However, they must be considered in the construction of the tree since they also have output events.

A ***causality path*** is a path in the tree from the environment node to the node with the locally inconsistent transitions. Every trigger in this path must fire if the locally inconsistent transitions are to be triggered. In Figure 3, there is only one causality path to transitions in state C, the upper branch. All causality paths to the node with the locally inconsistent transitions in every causality tree must be checked. The Backward Pass and the Forward Pass stages of the algorithm analyze each causality path separately in the context of the causality tree in which it resides.

6 Backward Pass: Predicate Positioning

Backward Pass begins by assuming that the inconsistency condition holds in the last microstep of the causality path in which the locally inconsistent transitions are triggered. It then determines the earliest microstep from which each predicate must hold if it is to hold at the last microstep. Let $\mathcal{P}$ be the set of predicates in the local inconsistency condition. In our running example from Section 2, the transitions c1→c2 and c1→c3 have guarding conditions that are both enabled under $in(b2) \wedge in(e2) \wedge in(a2) \wedge (m > 0)$ so $\mathcal{P}$= $\{in(b2), in(e2), in(a2), (m > 0)\}$.

Before proceeding, we introduce the notion of a ***suspect*** transition. A transition t is suspect if and only if it can cause a predicate p to become true.

$$Suspect(t,p) \equiv \begin{cases} s \in target\text{-}completion(t) & \text{if } p = in(\mathrm{s}) \\ s \in source\text{-}completion(t) & \text{if } p = \neg in(\mathrm{s}) \end{cases}$$

For a given predicate, we define its ***microstep assignment*** to be the microstep after which it can safely be assumed to be true if it is true at the last microstep. Each predicate is initially assigned to the last microstep. Backward pass then assigns each predicate p to the first preceding microstep that contains a suspect transition. If no suspect transition exists, p is assigned to microstep zero. This

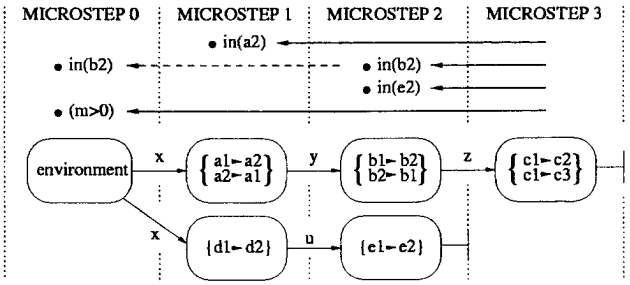

Fig. 4. The solid black lines show the result after the initial predicate assignments. The dotted line illustrates the reassignment of predicate $in(b2)$ to a previous microstep.

is a sound process since a predicate's truth value can only change at a microstep that contains a suspect transition.

The solid black lines in Figure 4 show the state after all of the predicates have been assigned. Predicate $in(a2)$ is assigned to microstep 1 since transition a1→a2 is suspect. Hence, $in(a2)$ can safely be assumed to be true in microsteps 2 and 3. Predicate $in(b2)$ can become true in microstep 2, so it can only be safely asserted in microstep 3. Likewise, predicate $in(e2)$ is assigned to microstep 2. Predicate $(m > 0)$ involves an input variable so it must have been true from the beginning of the superstep (microstep zero).

Next, we determine whether predicates can be reassigned to earlier microsteps. A predicate p is reassigned if all suspect transitions in its currently assigned microstep have guarding conditions that are unsatisfiable in the context of predicates assigned to previous microsteps. In such a case, p must have become true in an earlier microstep. It is thus reassigned to the next preceding microstep with a suspect transition. The reassignment of predicates is an iterative process since a reassignment may affect the microstep assignments of other predicates. This process is guaranteed to terminate since the number of preceding microsteps is finite and predicates can only be moved in one direction. The dotted line in Figure 4 shows the reassignment step. Predicate $in(b2)$ can be reassigned because the suspect transition b1→b2 has guarding condition $[(m < 0)]$ which is negated by predicate $(m > 0)$ assigned to a previous microstep.

Note that backward pass conservatively considers all of the nodes in the entire causality tree at each microstep, and not only the transitions triggered in the node in the causality path. In Figure 4, for instance, $in(e2)$ would be assigned to microstep zero if we do not consider nodes outside the causality path. This is not sound since transition e1→e2 may have made it true.

7 Forward Pass: Deriving a Contradiction

In the Forward Pass stage, we try to derive a contradiction based on the predicate assignments. Beginning with the first microstep, we look for transitions that (1) must be taken, and (2) falsify a predicate that was determined to be true at that microstep in the backward pass stage. We will refer to such transitions as ***violating*** transitions.

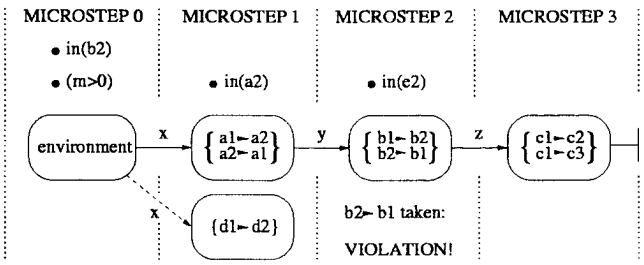

Fig. 5. Illustration of the Forward Pass stage. Predicate $in(a2)$ enables a transition from state $b2$ to $b1$ violating predicate $in(b2)$. (Although transition d1→d2 is not explicitly in the causality path, it is still considered since it is triggered by an event in the path.)

Unlike backward pass which examines the entire causality tree, forward pass looks only at the causality path and analyzes transitions triggered by trigger events in the path. This is because all trigger events in the causality path must fire in order for the locally inconsistent transitions to be ultimately triggered, and hence these are the only events that we can safely assume to have occurred.

The procedure for forward pass begins at microstep one and executes the following steps for each subsequent microstep:

1. Construct set ***MustHold*** that consists of all of the predicates assigned to previous microsteps. These predicates must be true in the current microstep in order for the local inconsistency condition to be valid. In Figure 5, ***MustHold*** for microstep 2 is $\{(m > 0), in(b2), in(a2)\}$.
2. Construct set ***EnabledT*** that consists of transitions triggered in the current microstep of the causality path and whose guarding conditions are enabled by asserting predicates in ***MustHold***. In Figure 5, ***EnabledT*** for microstep 2 is {b2→b1} since this transition has guarding condition $[in(a2)]$.
3. For each $p \in$ ***MustHold*** do
   - If p is of type $in(s)$: If there exists a transition t in ***EnabledT*** such that (1) the predicates in ***MustHold*** imply that we are in the source state of t, and (2) s is a member of $source\text{-}completion(t)$, then report violation.
   - If p is of type $\neg in(s)$: If (1) predicates in ***MustHold*** imply that we are in the parent state p of s and (2) ***EnabledT*** contains transitions from all child states of p other than s back to s, then report violation.

Note that if p is of type $\neg in(s)$, the fact that p, the parent state of s, is active guarantees that some child state of p other than s is active. Since we do not know which child state is active, we must ensure that there are enabled transitions from all child states of p other than s back to s.

In microstep 2 of Figure 5, transition b2→b1 must be taken since (1) predicate $in(b2)$ assigned to microstep 0 implies that we are in the source state of the transition, and (2) the guarding condition of the transition is satisfied by predicate $in(a2)$ assigned to microstep 1. Transition b2→b1 causes $b2$ to be inactive in microstep 3. This invalidates the local inconsistency condition. The constraint

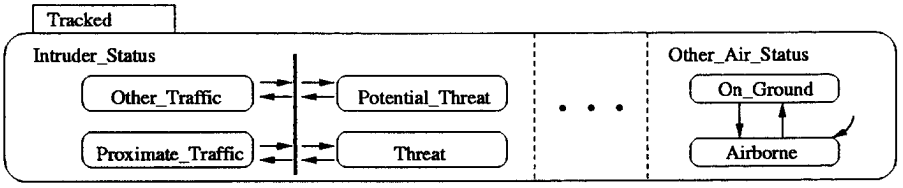

Fig. 6. Threat Detection Logic in CAS.

we can derive for this causality path is $[in(a2) \wedge (m > 0)] \Rightarrow \neg in(b2)$. Predicate $(m > 0)$ is included in the constraint only because it contributes to the reassignment of the violated predicate $in(b2)$. We have thus proven by contradiction that the local inconsistency condition can not hold for this causality path.

8 Deriving the Invariant

Since we must consider all causality paths to the locally inconsistent transitions, a violation must be found for each path. Otherwise, no invariant can be safely formulated. Suppose we have the following violation constraints for n causality paths: $(P_1 \Rightarrow \neg p_1, P_2 \Rightarrow \neg p_2, ..., P_n \Rightarrow \neg p_n)$, where P_i denotes the conjunction of predicates which once asserted, guarantees the negation of predicate p_i for the *ith* causality path. The invariant is then the disjunction of the n violation constraints: $(P_1 \wedge P_2 \wedge ... \wedge P_n) \Rightarrow (\neg p_1 \vee \neg p_2 \vee ... \vee \neg p_n)$. This invariant applies only when the trigger event to the locally inconsistent transitions occurs.

9 Application to TCAS II

Our method was applied to a part of the TCAS II ([air] Traffic alert and Collision Avoidance System) specification version 6.04A written in RSML. It was used to supplement consistency checking in the Collision Avoidance Subsystem (CAS). CAS models intruding aircraft and classifies them as one of *Threat*, *Potential_Threat*, *Proximate_Traffic*, or *Other_Traffic*. Figure 6 shows a part of CAS that models the intruding aircraft. The transition bar in state *Intruder_Status* is shorthand notation for transitions between any two states. State *Other_Air_Status* models the intruding aircraft as either being close to the ground (state *On_Ground*), or airborne (hence more of a threat).

Using the Stanford Validity Checker (SVC), we discovered a local inconsistency condition for transitions *Proximate_Traffic* to *Potential_Threat* and *Proximate_Traffic* to *Other_Traffic*. It includes, in part, the predicates in(*On_Ground*) and $\neg$*Other_Alt_Reporting*. This means that the intruding aircraft is not reporting altitude but it is classified as being close to the ground.

By applying the analysis described in this paper, our tool generated the invariant $\neg$*Other_Alt_Reporting* $\Rightarrow \neg$in(*On_Ground*). This invariant, as well as another one that was critical in consistency checking the specification, were found in no more than two seconds using our prototype tool written in LISP. However, the runtime is an underestimate since we did not fully expand all causality trees; the entire specification was not available in an electronic format. The parts that were left unspecified were nevertheless determined to be irrelevant to our analysis.

10 Discussion

We analyze a conservative approximation of the execution that statically determines all possible changes that can occur in the system at any given time with any given input. Since the approximation has less information than the original specification, we may overlook properties that are in fact true of the specification. On the other hand, the limited size of the approximation makes it computationally feasible to analyze. The algorithm has been integrated into a general consistency checking prototype tool. We expect to extend it with other static analysis tools as they become available.

Acknowledgements

We have benefitted greatly from the collaboration with Mats Heimdahl, University of Minnesota, who has provided us with insights into RSML in general and comments to an earlier draft of the paper.

References

1. R.J. Anderson, P.Beame, S. Burns, W. Chan, F. Modugno, Notkin D, and J.D. Reese. Model checking large software specifications. In D. Garlan, editor, *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (SIGSOFT'96)*, pages 156–166, October 1996.
2. C. Barrett D.L. Dill and J. Levitt. Validity checking for combinations of theories with equality. In M. Srivas and A. Camilleri, editors, *Formal Methods in Computer Aided Design (FMCAD)*, number 1166 in Lecture Notes in Computer Science, pages 197–201. Springer-Verlag, November 1996.
3. S. Gerhart, D. Craigen, and T. Ralston. Formal methods reality check: Industrial usage. *IEEE Transactions on Software Engineering*, 21(2):90–98, February 1995.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
5. D. Harel and A. Pnueli. On the development of reactive systems. In K.R. Apt, editor, *Logics and Models of Conc. Systems*, pages 477–498. Springer-Verlag, 1985.
6. M. P.E. Heimdahl and N.G. Leveson. Completeness and consistency analysis of state-based requirements. *IEEE TSE*, 22(6):363–377, June 1996.
7. C. L. Heitmeyer, R. D. Jeffords, and B. G. Labaw. Automated consistency checking of requirements specifications. *TOSEM*, 5(3):231–261, July 1996.
8. D.N. Hoover and Zewei Chen. Tablewise, a decision table tool. In J. Rushby, editor, *Proceedings of 10th Annual Conference on Computer Assurance (COMPASS '95)*, pages 97–108, Gaithersburg, MD, USA, June 1995. IEEE.
9. M. S. Jaffe, N. G. Leveson, M. P.E. Heimdahl, and B. Melhart. Software requirements analysis for real-time process-control systems. *IEEE Transactions on Software Engineering*, 17(3):241–258, March 1991.
10. N.G. Leveson, M. P.E. Heimdahl, H. Hildreth, and J.D. Reese. Requirements specification for process control systems. *IEEE Transactions on Software Engineering*, 20(9):694–707, September 1994.
11. D. Y.W. Park, J.U. Skakkebæk, M. P.E. Heimdahl, B.J. Czerny, and D.L. Dill. Checking properties of safety critical specifications using efficient decision procedures. In *Formal Methods in Software Practice*, pages 34–43. ACM Press, 1998.
12. D. L. Parnas, G. J. K. Asmis, and J. Madey. Assessment of safety-critical software in nuclear power plants. *Nuclear Safety*, 32(2):189–198, April–June 1991.